A linear tree matching algorithm

AUTHOR(S):

Kojima, Keiji

CITATION:

Kojima, Keiji. A linear tree matching algorithm. 数理解析研究所講究録 1983, 482: 129-144

ISSUE DATE:

1983-03

URL:

http://hdl.handle.net/2433/103407

数理解析研究所講究録
482巻 1983年 129-144

「ソフトウエア科学、工学における数理的方法」
研 究 集 会
京都大学数理解析研究所
1982年5月24日 ～ 26日

A linear tree matching algorithm

by

Keiji Kojima

Central Research Laboratory, Hitachi, Ltd.

Kokubunji, Tokyo 185, Japan

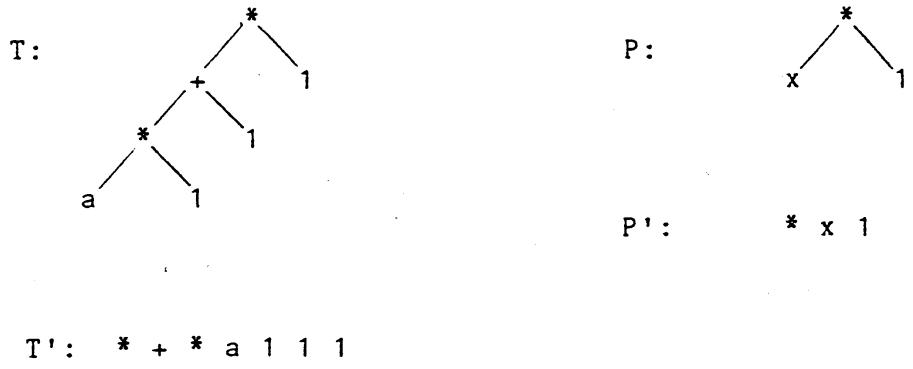
Abstract

An algorithm is presented which searches all occurrences of a given complete binary tree in another, in running time proportional to the sum of the numbers of their nodes. The algorithm is essentially an application of Knuth-Morris-Pratt's string matching algorithm. An extension to more general tree structures is also described.

1. Introduction

Pattern matching for typical data structures (string, tree, graph etc.) plays an important role in symbolic manipulations [1]. For string matching problems, a number of algorithms have been introduced ([2],[3],[4],[5]). Knuth-Morris-Pratt's string matching algorithm (KMP) [3] is noted among others as the first linear time algorithm. This paper introduces a linear pattern matching algorithm for binary trees, whose basic idea is derived from KMP.

Tree matching algorithm have wide-ranged applications in information processing e.g. automatic algebraic simplification and logical deduction. Figure 1.1 illustrates the example of the use of the tree matching algorithm.



The binary tree T represents the algebraic expression (a*1+1)*1 which can be simplified to a+1 using the formula x*1=x. In order to perform this simplification, it is necessary to find all occurences of P in T regarding x as a special node that matches to any node of T. An ovious solution is to apply a string matching algorithm by transforming trees to strings. In fig.1.1, T' and P' are strings which are obtained from T and P by preorder traversing. In fact, *a1, which is an occurrence of P', will be found if we apply a string matching algorithm to T' and P'. Unfortunately, however, no other occurence of pattern P in T can

be detected from P' or T'. Such defect is not particular to preorder ordering. It can be easily proved that no matter what representation may be used, extra informations are required in order to disaround this defect. Besides the ovious loss of time which is consumed for tree-string transformations, this fact shows that use of string matching algorithm is not the best solution for the tree matching problem.

The tree matching algorithm which is presented in this paper works on tree structures directly. In KMP, string matching is considered as placing the pattern string (say x) over the text string (say y) and sliding x to the right. KMP slides x over y as far as possible making use of the information which is constructed by the analysis of x. The analysis of x requires $O(|x|)$ time (i.e. time proportional to the length of x) and the sliding x over y requires $O(|y|)$ time. Therefore KMP requires $O(|x|+|y|)$ time on a whole. In order to attempt to apply the idea of KMP to tree structures, following problems must be solved.

(i) What information is necessary in order to slide a pattern tree over a text tree efficiently ?

(ii) Is the information able to be constructed in linear time ?

The tree matching algorithm which is described in this paper solves these problems in the case that the pattern tree is complete binary. The algorithm has two stages just as KMP : pattern analysis and matching between text and pattern.

In Section 2, we introduce the basic data representations as the basis for the discussion of the later section. In Section 3, a linear time algorithm for the analysis of the pattern tree is described. In Section 4, we describe the matching algorithm whose running time is also linear. In Section 5, the matching algorithm is extended to cover text trees which are not necessarily complete.

2. Data Structures

In this section, we introduce the basic data structures and the operations on them. Given a complete binary tree with 2^h-1 ($h>0$) nodes, we represent it by an array T in the following manner.

(i) the root of the tree is stored in T[1].

(ii) the left son and right son of the node which is stored in T[i], are stored in T[2i] and T[2i+1] respectively.

That is, the nodes of the tree are stored in level-first order in array. So hereafter let the nodes of a complete binary tree be denoted by their level-first numbers.

We define "compact subtree" which is a suitable unit to our discussion.

Definition 2.1

Let T be a complete binary tree. A compact subtree T(i,j)

is the tree whose nodes are descendants of i and numbered less than or equal to j. The node i is the root of T(i,j). The node j is called the bottom of T(i,j). |T(i,j)| represents the number of the nodes of T(i,j). (fig.2.1)

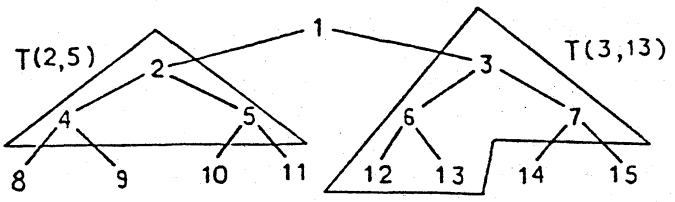


fig. 2.1

Using def.2.1, our tree matching problem can be formally described as follows.

Problem

Let T and P compelte binary trees. T is called a text tree and P is called a pattern tree. Construct an O(|T|+|P|) algorithm that searches all compact subtrees of T which is equal to P.

We define three operations which give characteristic positions on compact subtrees.

Definition 2.2

(i) $left(i,j) = 2^{\lfloor \log j \rfloor - \lfloor \log i \rfloor} * i \quad (i,j \neq 0)$

(ii) $right(i,j) = 2^{\lfloor \log j \rfloor - \lfloor \log i \rfloor} * (i + 1) - 1 \quad (i,j \neq 0)$

(iii) $fwd(i,j) =$ if $left(i,j) \leq j < right(i,j)$ then $j + 1$ else $2*left(i,j) \quad (i,j \neq 0)$

These functions are also defined on i=0 as left(0,j)=j, right(0,j)=j and fwd(0,j)=j. (fig.2.2)

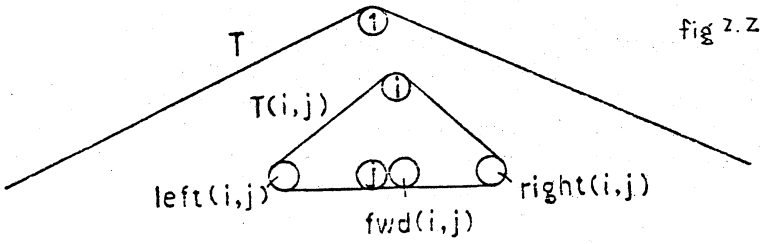


fig 2.2

Proposition 2.1

Let T be a complete binary tree and T(i,j) be a compact subtree of T.
(i) If T(i,k) is the minimal complete compact subtree of T which contains T(i,j), then left(i,j) is the leftmost leaf and right(i,j) is the rightmost leaf of T(i,k).
(ii) T(i,fwd(i,j)) is a compact subtree which contains T(i,j) and |T(i,fwd(i,j))|=|T(i,j)| + 1.

Corollary

(i) left(i,j)=j iff T(i,k) is a complete compact subtree where j=fwd(i,k).
(ii) right(i,j)=j iff T(i,j) is a complete compact subtree.

Two auxiliary functions are introduced by the following definition.

Definition 2.3

(i) $trans(i,j) = j - 2^{\lfloor \log j \rfloor - \lfloor \log i \rfloor} * (i-1)$ $(i,j \neq 0)$
(ii) $trans^{-1}(j,1) = (1-j) * 2^{-\lfloor \log i \rfloor} + 1$ $(j,1 \neq 0)$

It can be easily proved that trans(i,j)=|T(i,j)|. trans is an inverse of trans in the sense that $trans^{-1}(j,trans(i,j))=i$.

3. Analysis of pattern tree

In this section, we describe the analyzing algorithm which produces information used to speed-up the matching procedure. This algorithm takes a linear time for the size (number of nodes) of pattern tree. We assume that pattern tree is stored in array P and that its size is m. First of all, we define five characteristic compact subtrees. These trees supply all necessary informations for the linear matching.

Definition 3.1

Candidate tree (C-tree) of the node j is the maximal compact subtree P(i,k) that satisfies following conditions.
(i) $i \neq 1$
(ii) $j = fwd(i,k)$
(iii) $P(i,k) = P(1,trans(i,k))$
If there is no such a compact subtree, C-tree for j th node of T is an empty tree. This also holds for def.3.1 to def.3.5. Notice that C-tree for j is unique because there always exists the maximal tree among compact subtrees with the same bottom.

Definition 3.2

Success tree (S-tree) of the node j is the maximal compact subtree P(i,j) ($i \neq 1$) such that P(i,j) is equal to P(1,trans(i,j)).

Definition 3.3

Failure tree (F-tree) of the node j is the maximal compact subtree P(i,j) (i ≠ 1) that satisfies following conditions.
(i) P(i,k) = P(1,trans(i,k)) where j = fwd(i,k)
(ii) P[j) = P[trans(i,j))

Definition 3.4

Right tree (R-tree) of the node j is the maximal compact subtree P(i,k) (i ≠ 1 , j = fwd(i,k)) such that P(i,j) is complete and equal to P(1,trans(i,j)).

Definition 3.5

Left tree (L-tree) of the node j is the maximal comapct subtree P(i,j) (i ≠ 1) such that P(i,j) is complete and equal to P(1,trans(i,j)).

We define functions which return the root of C,S,F,L or R-tree of a given node.

Definition 3.6

Let X represent 'C','S','F','L' or 'R'. Then,
X-root(j)=i iff i is the root of X-tree for j.
If the corresponding tree is an empty one, the value of function is 0. For example, C-root(j)=i if i is the root of C-tree for j th node and C-root(j)=0 if C-tree for j th node is empty.

We define one more convenient function named 'next'.

definition 3.7

Let X be 'C','S','F','L' or 'R'. Then,
next(X,j) = trans(X-root(j),j))
$next^{(k)}$ (X,j) = next($next^{(k-1)}$(X,j))

Our next task is to construct C,S,F,L and R-tree in a linear time. This can be achieved by utilizing the inductive relations among these trees. Firstly, we show this inductive relations. The following five propositions show that we are able to find S-,F-,L- and R-tree of the node j if we know C-tree of the node j and these four trees of nodes whose numbers are less than j.

Proposition 3.1

Assume that C-tree for j (j>1) is an empty tree. (i.e. C-root(j)=0)
(i) If P[j]=P[1], then S-root(j)=j, F-root(j)=0, L-root(j)=j and R-root(j)=0.
(ii) Otherwise, S-root(j)=0, F-root(j)=j, L-root(j)=0 and R-root(j)=0.

Proposition 3.2

Assume that C-root(j)=i (i≠0) and that P[j]=P[trans(i,j)].
(i) S-root(j)=i
(ii) F-root(j)=trans^{-1}(j,next(F,trans(i,j)))

Proposition 3.3

Assume that C-root(j)=i (i≠0). If left(i,j)=j, then R-root(j)=i else R-root(j)=trans^{-1}(j,next(R,trans(i,j))).

Proposition 3.4

Let i=S-root(j). If right(i,j)=j then L-root(j)=i. Otherwise, L-root(j)=trans^{-1}(j,next(L,trans(i,j))).

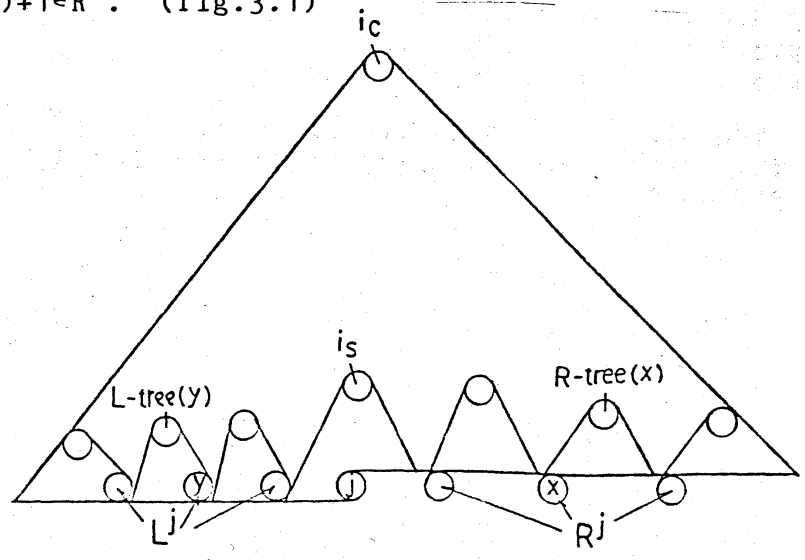
Proposition 3.5

Assume that C-root(j)=ic (ic≠0) and that P[j]≠P[trans(ic,j)].
(i) Let m be the smallest integer which satisfies P[j]=P[next$^{(r)}$(F,j')] or next$^{(r)}$(F,j')=0, where j'=trans(ic,j). Then S-root(j)=trans^{-1}(j,next(m)(F,j')).
(ii) F-root(j)=ic

If C,S,F,L and R-tree of nodes whose numbers are less than or equal to j, some C-trees of the nodes whose numbers are greater than j are known. The following two definitions and three propositions show this.

Definition 3.8

Let is=S-root(j) and ic=C-root(j). R^j is a set nodes, which is constructed as follows.
(i) If right(is,j) < right(ic,j), then right(is,j)+1∈R^j.
(ii) If x∈R^j and right(R-root(x),x) < right(ic,j), then right(R-root(x),x)+1∈R^j. (fig.3.1)



Definition 3.9

Let is=S-root(j) and ic=C-root(j). Then Λ^{L^j} is a set of nodes and constructed as follwing.
(i) If left(is,j) > left(ic,j), then left(is,j)-1$\in L^j$.
(ii) If $x \in L^j$ and left(L-root(x),x) > left(ic,j), then left(L-root(x),x)-1$\in L^j$.

Proposition 3.6

Let is=S-root(j). If is=0, then C-root(2j)=0 and C-root(2j+1)=0. Otherwise, C-root(fwd(is,j))=is.

Proposition 3.7

Let $x \in R^j$ and ic=C-root(j). Then,
C-root(x)=$trans^{-1}$(x,next(R,trans(ic,x))).

Proposition 3.8

Let $x \in L^j$ and ic=C-root(j). If L-root(x)=0, then C-root(2x)=0 and C-root(2x+1)=0. Otherwise, C-root(fwd(L-root(x),x))=L-root(x).

Now we are able to establish the pattern analysis algorithm which constructs C-,S-,F-,R- and L-tree for a complete binary tree T. The algorithm is based on the inductive relations of C,S,F,R,and L-tree which are shown by prop.3.1 to prop.3.8. That is, each part of the algorithm corresponds to prop.3.1 to prop.3.8 strictly. This correspondence is as follows.

prop.3.1 : line 6,7,8
prop.3.2 : line 11
prop.3.3 : line 16,17
prop.3.4 : line 18,19,20
prop.3.5 : line 12,13,14,15
prop.3.6 : line 22,23
prop.3.7 : line 24,25,26,27
prop.3.8 : line 28,29,30,31,32

The algorithm uses five array c,s,f,l and r corresponding to the five tree. For example, f[j]=F-root(j) after the algorithm was executed.

Algorithm 3.1

```
1       (c[1],s[1],f[1],r[1],l[1]):=(0,0,0,0,0);
2       (c[2],c[3]):=(0,0);
3       for j from 2 to m do
4         ic:=c[j];
5         if ic=0
            then
6             if P[j]=P[1]
7               then (s[j],l[j]):=(j,j); (f[j],r[j]):=(0,0)
8               else f[j]:=j; (s[j],r[j],l[j]):=(0,0,0)
              end if
            else
9             j':=trans(ic,j);
10            if P[j]=P[j']
11              then s[j]:=ic; f[j]:=trans^-1(j,next(F,j'))
12              else x:=j'; f[j]:=ic;
13                   until P[x]=P[j ] or x=0 do
14                     x:=next(F,x)
                     end until;
15                   s[j]:=trans^-1(j,x)
              end if;
16            if left(ic,j)=j then r[j]:=ic
17                            else r[j]:=trans^-1(j,next(R,j'))
              end if;
18            if right(ic,j)=j
19              then l[j]:=s[j]
20               else l[j]:=trans^-1(j,next(L,trans(s[j],j)))
              end if
          end if;
21        is:=s[j];
22        if is=0 then (c[2j],c[2j+1]):=(0,0)
23                else c[fwd(is,j)]:=is
          end if;
24        x:=right(is,j)+1;
25        while x=<right(ic,j) do
26          c[x]:=trans^-1(x,next(R,trans(ic,x)));
27          x:=right(c[x],x])+1
          end while;
28        x:=left(is,j)-1;
29        while x>=left(ic,j) do
30          if l[x]=0 then (c[2x],c[2x+1]):=(0,0)
31                    else c[fwd(l[x],x)]:=l[x]
            end if;
32          x:=left(l[x],x)-1
          end while
        end for
```

The correctness and complexity of algorithm3.1 are described by the following theorems.

Theorem 3.1

Algorithm 3.1 computes C-tree, S-tree, F-tree, R-tree and L-tree correctly.

Proof of Theorem 3.1

Each tree of the node 1 of T is correctly computed at line 1 of algorithm 3.1 by definition of each tree. Assume that c[j]=C-root(j) at line 4. If c[j]=0, then S-root(j), F-root(j), R-root(j) and L-root(j) are correctly computed by prop.3.1. So, let c[j]=0. If P[j]=P[trans(C-root(j),j)], then S-root(j) and F-root(j) are correctly computed at line 11 by prop.3.2, otherwise they are also correctly computed at line 12-15 by prop.3.5. L-root(j) and R-root(j) are correctly computed at line 16-17 and line 18-19 by prop.3.4 and prop.3.3 respectively. Therfore it is sufficient to prove this theorem if we show c[j]=C-root(j) at line 4. To show this, we provide the loop invariant (*) at line 4.

(*) $\bigwedge_{k=1}^{l}$ c[jk] = C-root(jk)

where j1=j, jk=right(c[jk],jk)+1 (1<k<l+1)

and jl+1=2left(c[j1],j1). (fig.3.2)

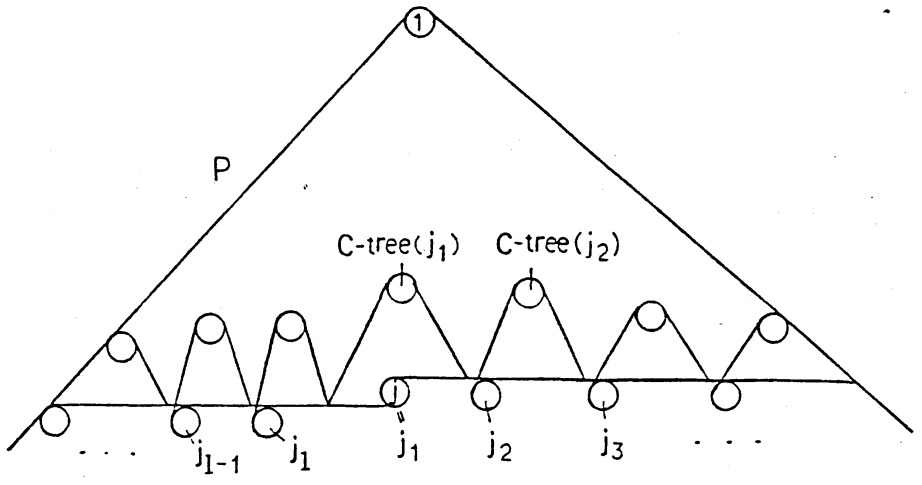


fig.3.2

When j=2, (*) becomes c[2]=C-root(2) c[3]=C-root(3) and this is true since C-root(2)=C-root(3)=0 by def.3.1 and prop.3.6 and c[2]=c[3]=0 at line 2. Now assume that (*) is true for j (j>1). If C-root(j)=S-root(j)=0 and j<right(C-root(j),j), then j+1=fwd(C-root(j),j). By prop.3.6 C-root(j+1)=C-root(j). Hence

(*) is still true after one for-loop traversal. If C-root(j)=S-root(j)=0 and j=right(C-root(j),j), then j+1=j2 and jl+1=fwd(C-root(j),j)=2left(C[j2],j2). Since C-root(j2)=c[j2]by hypothesis and C-root(right(C-root(jl),jl)+1)=c[right(C-root(jl),jl)+1], c[fwd(C-root(j),j)] is correctly assigned at line 23 by prop.3.6. Hence (*) is also true after one for-loop traversal. If C-root(j)=S-root(j)=0, then j+1=j2,2j=jl+1, 2j+1=right(C-root(jl+1),jl+1)+1 and right(c[2j+1],2j+1)+1=2left(c[j2],j2). Since C-root(j+1)=c[j+1], C-root(2j)=c[2j] and C-root(2j+1)=c[2j+1] by hypothesis and prop.3.6, (*) is true in this case. If C-root(j)=S-root(j), then R^j and L^j can not be empty. From the construction of R^j and L^j, however, (*) is clealy true. Therefore we have established (*) as the loop invariant for for-loop. From (*), c[j]=C-root(j) at line 4 especially. Hence algorithm 3.1 correctly computes each tree.

Theorem 3.2

Algorithm 3.1 requires O(m) time.

Proof of Theorem 3.1

The algorithm has three loop at line 13, line 25 and line 29 on which the time complexity depends. Let us define two functions G and H in order to evaluate the cost of these three loops.

$$G(j) = \sum_{k=1}^{l} |\text{C-tree for } jk|$$

$$H(j) = l$$

where j1=j, jk=right(c[jk-1],jk-1)+1 $(1 \le k < l+1)$

jl+1=2left(c[j1],j1).

Notice that c[jk]=C-root(jk) at line 4 by the loop invariant in proof of theorem 3.1. Assume that x:=next(F,x) is executed p(j) times at line 14. Then G(j) decreases by p(j)-1 at least because $p(j) < C\text{-root}(j) - S\text{-root}(j)$ and j becomes an element of C-tree for fwd(S-root(j),j) if S-root(j)≠0. Therfore $G(j+1) \le G(j)-p(j)+1$. Hence $\sum_{j=1}^{m} p(j) \le G(1)-G(m+1)+m \le m$ since G(1)=0 and G(m+1)≥0. That is, line 14 is executed at most m times during the execution of algorithm 3.1. On the other hand, if the while loops at line 25 and 29 are repeated q(j) times, then $H(j+1) \ge H(j)+q(j)$ since at least q(j) new C-trees are added. On a whole,

$q(j) \le H(m+1)-H(1) \le m$ since $H(m+1) \le m+1$ and H(1)=1. Therfore the total cost of the loops at line 13, 25 and 29 is O(m). Hence algorithm 3.1 requires O(m) time.

4.Matching algorithm

We are going to construct a linear pattern matching algorithm for complete binary trees in this section. We assume that text tree (size n) is stored in array T and pattern tree (size m) is stored in array P. For pattern tree P, we have already its F-tree, L-tree and R-tree in array f,l and r respectively by Algorithm 3.1. C-tree and S-tree are not used in the matching algorithm which we will construct. First of all. we revise the definitions of C-trees and S-trees.

Definition 4.1

C-tree for j th node of T is the maximal compact subtree T(i,k) which satisfies following conditions.
(i) j = fwd (i,k)
(ii) T(i,k) = P(1,trans(i,k))

Definition 4.2

S-tree for j th node of T is the maximal comapct subtree T(i,j) which is equal to P(1,trans(i,j)) where trans(i,j)≠m

Since def.3.1 and def.3.2 are the special case that T=P and i=1 in def.4.1 and def.4.2, the revised versions of C-tree and S-tree are the extensions of old ones. From hereafter, we adopt def.4.1 and def.4.2 as the definitions of them. Because we do not utilize C-tree and S-tree of P which are constructed by algorithm 3.1 at all in this section, this causes no confusion. By def.4.1, constructing C-tree for every node of T is equivalent to solving our pattern matching problem. That is, we are searching C-tree T(i,k) such that |T(i,j)|=m and T[j]=P[m] with j=fwd(i,k). From this point of view, algorithm 3.1 is considered to be a matching algorithm between pattern tree and itself by regarding T as P in def.3.1 and def.3.2. Using def.4.1 and def.4.2, prop3.1,3.2 and 3.5 holds with minor changes.

Proposition 4.1

Assume that C-root(j)=0. If T[j]=P[1], then S-root(j)=j. Otherwise, S-root(j)=0.

Proposition 4.2

Assume that C-root(j)=i (i≠0) and that T[j]=P[trans(i,j)]. If trans(i,j)≠m, then S-root(j)=i. Otherwise, S-root(j)=$trans^{-1}$(j,next(L,trans(i,j))).

Proposition 4.3

Assume that C-root(j)=i (i≠0) and that T[j]≠P[trans(i,j)]. Then, S-root(j)=$trans^{-1}$(j,$next^{(r)}$(F,j')) where j'=trans(i,j) and l is the smallest integer which satisfies T[j]=P[$next^{(r)}$(F,j')] or $next^{(r)}$(F,j')=0.

Prop.3.6, 3.7 and 3.8 hold true without any change.
Now we show the matching algorithm. This algorithm searches all T(i,j) that is equal to P and prints pair (i,j). As we have mentioned, the matching algorithm is essentially the same as algorithm 3.1. The correspondence between propositions and lines is as follows.

prop.4.1 : line 5
prop.4.2 : line 8
prop.4.3 : line 9,10,11
prop.3.6 : line 14,16
prop.3.7 : line 17,18,19,20
prop.3.8 : line 21,22,23,24,25,26

Algorithm 4.1

```
1     c[1]:=0;
2     for j from 1 to n do
3       ic:=c[j];
4       if ic=0
          then
5           if T[j]=P[1] then s[j]:=j else s[j]:=0 end if
          else
6           j'=trans(ic,j);
7           if T[j]=P[j']
8             then s[j]:=ic; if j'=m then print(ic,j)
9             else x:=j';
10                 until T[j]=P[x] or x=0 do
11                   x:=next(F,x)
                   end until;
12                 s[j]:=trans^-1(j,x)
            end if;
        end if;
13      is:=s[j];
14      if is=0 then (c[2j],c[2j+1]):=(0,0)
                else
15                if j'=m then is:=trans^-1(j,next(L,j')) end if;
16                c[fwd(is,j)]:=is
        end if;
17      x:=right(is,j)+1;
18      while x=<right(ic,j) do
19        c[x]:=trans^-1(x,next(R,trans(ic,x)));
20        x:=right(c[x],x)+1
        end while;
21      x:=left(is,j)-1;
22      while x>=left(ic,j) do
23        y:=trans^-1(x,next(L,trans(ic,x)));
24        if y=0 then (c[2x],c[2x+1]):=(0,0)
25               else c[fwd(y,x)]:=y
          end if;
26        x:=left(c[x],x)-1
        end while
      end for
```

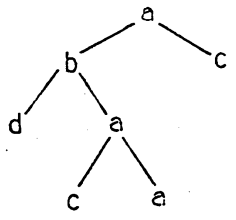
Theorem 4.1

Algorithm 4.1 searches all occurence of T(i,j) such that T(i,j)=P in O(n) time.

We are able to prove this theorem in quite similar manner supplying the same for-loop invariant and the cost evaluation functions as proof of Theorem 3.1 and Theorem 3.2. We omit the detail.

5. Extension

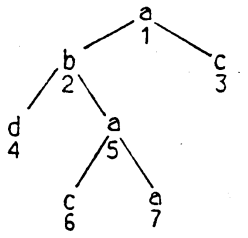
Algorithm 4.1 can be extended to cover text trees which are not necessarily complete. Assume that a text tree T with n nodes is stored in array text in level-first order. Lson, rson and parent are arrays such taht lson[k], rson[k] and parent[k] are the left son, right son and parent of node k respectively. (Fig.5.1)



T

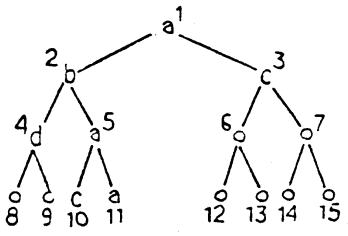
k	1	2	3	4	5	6	7
text	a	b	c	d	a	c	a
parent	0	1	1	2	2	5	5
lson	2	4	0	0	6	0	0
rson	3	5	0	0	7	0	0

T is embedded to a complete binary tree T' by two functions f:T->T' and f':T'->T. That is, for node k of T, f(k) gives the corresponding node of T' and f' is the inverse of f. (fig.5.2)



T

k	1	2	3	4	5	6	7
f	1	2	3	4	5	10	11



T'

k	1	2	3	4	5	10	11
f'	1	2	3	4	5	6	7

The table of f and f' can be constructed in O(n) time in the ovious manner. It is possible to calculate f' for nodes which don not appear in the table but their parents do.

$$f'(k) = \text{if } k \text{ is even then } lson[f'(k/2)] \text{ else } rson[f'(k-1/2)]$$

Note that the number of the nodes of T' for which f' is defined is O(n). Now assume that the construction of C-tree for T and P proceeds upto j th node of T and that C-root(j)=ic. Then, using f, T(ic,j) is translated to T'(f(ic),f(j)). (fig.5.3) That is, f(j) and f(ic) are used instead of j and ic in algorithm 4.1.

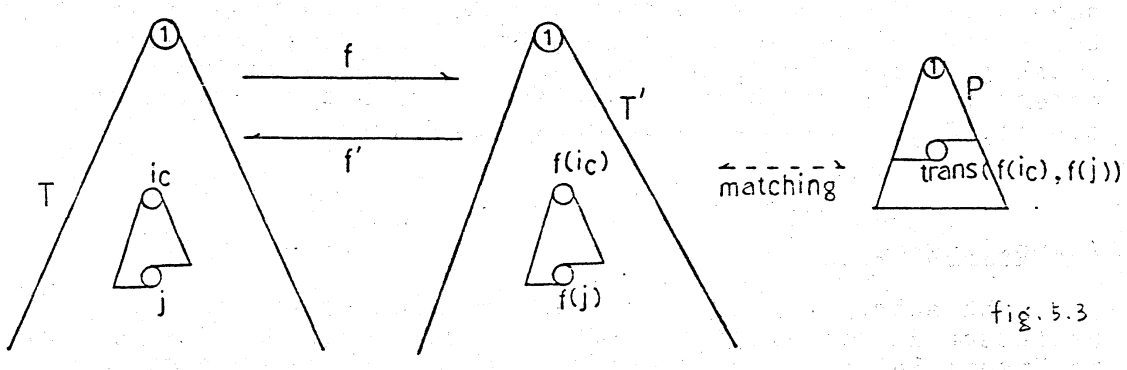
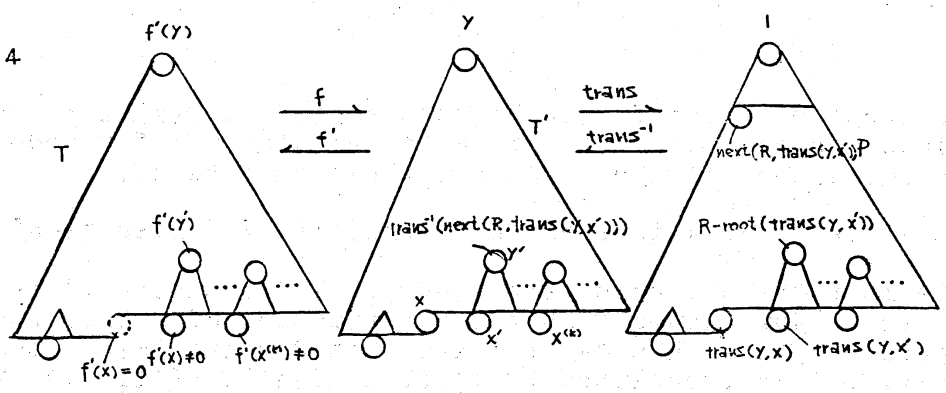


fig. 5.3

Since T' is complete, algorithm 4.1 correctly works for T'(f(ic),f(j)) and decides C-trees for appropriate nodes of T'. Let us assume that C-root(x)=y for nodes x and y of T'. This time T'(y,x) is translated to T(f'(y), f'(x)). That is, let C-root(f'(x))=f'(y) in T.

There is a possibility, however, that the corresponding node to x may not exist in T (i.e. f'(x)=0. Notice that f' is defined for x since f'(parent[x]) is not zero in algorithm 4.1.) In this case the extended algorithm searches the node x' of T' such that $x<x' \leq right(y,x)$ and $f'(x') \neq 0$. For such x', C-root(f'(x))=f'(y') in T where $y'=trans^{-1}(x',next(R,trans(y,x')))$. If there remains the node x'' such that $right(y',x')<x'' \leq right(y,x)$ and $f'(x'') \neq 0$, then C-root(f'(x''))=f'(y'') where $y''=trans^{-1}(x'',next(R,trans(y,x'')))$. This process continues untill there remains no such x . A similar process is performed to the left of x using appropriate L-trees and R-trees.(fig.5.4)

fig 5.4



After these processes, the construction of C-tree successfully proceeds to j+1 th node of T since all necessary C-trees have been decided and quite a similar loop invariant which is used in the proof of Theorem 3.1 also holds. The time complexity of the extended algorithm is O(n) since we have only accessed to the nodes of T' on which f' is defined. The space complexity is clearly O(n) from the above data structures.

Acknowledgements

The author wishes to express his deep appreciation to Professor Reiji Nakajima for his helpful advices. He also thanks to Etsuya Shibayama and Tatsuya Hagino for valuable discussions with them.

References

[1] Donald E.Knuth, Fundamental Algorithm, The art of Computer Programming, Vol.1, Addison-Wesley, Reading,Mass., 1968; 2nd edition 1973.
[2] Malcom C.Harrison, Implementation of the substring test by hashing, Comm.ACM,14 (1971), pp 777-779.
[3] D.E.Knuth, J.H.Morris,Jr., V.R.Pratt, Fast Pattern Matching in Strings, SIAM J. of Computer, Vol.6, No.2, June 1977.
[4] Alfred .V.Aho and Margaret J.Corasick, Efficient string matching: An aid to bibliographic search, Comm.ACM, 18 (1975), pp.333-340.
[5] R.S.Boyer and J.S.Moore, A first string searching algorithm, Comm.ACM, Vol.20, No.2, Oct. 1977.